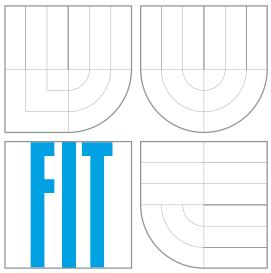
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

STATICKÁ ANALÝZA PROGRAMŮ V C VE SPARSE A PŘÍBUZNÝCH NÁSTROJÍCH

STATIC ANALYSIS OF C PROGRAMS IN SPARSE AND SIMILAR TOOLS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MARTIN NAGY

VEDOUCÍ PRÁCE
SUPERVISOR

doc. Ing. TOMÁŠ VOJNAR, Ph.D.

BRNO 2009

Abstrakt

Softwarová verifikace se postupně stává čím dál tím více důležitou součástí vývoje. Jejím cílem je zajištění kvality výstupního produktu. Navzdory tomuto ale problém psaní dobrých nástrojů pro statickou analýzu často spočívá v nedostatku dobrého front-endu překladače. Tato práce se pokouší analyzovat a zdokumentovat existující nástroj, zvaný Sparse, aby byli vědečtí pracovnící vyzbrojeni stabilním řešením, které jim umožní vyvíjet jejich analyzátory. V neposlední řadě je taky diskutovaný projekt Mygcc a jeho nový přístup k integraci se stávajícími překladači.

Abstract

Software verification is steadily becoming important for software developers and companies to ensure software quality. However, the problem of writing a good static code analysis tool often stems from the lack of a good compiler front-end. To solve this problem, we try to analyse and document an existing tool called Sparse to empower software verification researchers with a ready, stable solution for their projects. Additionally, we also talk about Mygcc and it's new approach to integrate with existing compilers.

Klíčová slova

Sparse, Mygcc, front-end překladače, statická analýza kódu, neparsované porovnání vzorů

Keywords

Sparse, Mygcc, compiler front-end, static code analysis, unparsed pattern matching

Citace

Martin Nagy: Static Analysis of C Programs in Sparse and Similar Tools, bakalářská práce, Brno, FIT VUT v Brně, 2009

Static Analysis of C Programs in Sparse and Similar Tools

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana doc. Tomáše Vojnara. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Martin Nagy
May 20, 2009

Poděkování

Rád bych poděkoval mému vedoucímu doc. Tomáši Vojnarovy za odborné vedení, poskytnuté rady a čas, který mi při tvorbě práce věnoval.

Contents

Chapter 1

Introduction

In formal verification, static source code analysis refers to the methodology of analyzing software by converting the software's source code into an abstract representation that is suitable for manipulation by various analysis methods. These methods can deduce various properties about the program in question.

In this work, we are going to take a look at two tools: Sparse and Mygcc. We will study their capabilities and usefulness for the purpose of static code analysis. In the case of Sparse, we will also study it's compiler front-end that parses C source code and transforms it into a syntax tree. The front-end (as Sparse itself) is heavily undocumented, which is why we try to summarize its usage in a concise manner to help other people with writing a static source code analysis tool (or any other application needing a compiler front-end). The Mygcc tool is an attempt to incorporate static code analysis methods into the gcc compiler. While the project is, for now, still not successful and widely adopted, we will show why its potential success could have a profound impact on all software developers.

In the following chapters, we will introduce the Sparse tool that is used to find defects in the Linux kernel. We will also describe the Sparse library – a compiler front-end supporting standard ISO C and many GNU C extensions. It is the library that has many interesting capabilities and empowers the tool itself. After Sparse, we will describe Mygcc and its intriguing method that enables it to do very powerful code analysis. Even if not perfect and sound, it is very elegant, fast and easy to incorporate into existing compilers. We will then describe our own contribution and experience we gained by using these tools and learning about them.

Chapter 2

Sparse

Sparse is a semantic parser for C source files written by Linus Torvalds in 2003. It is a lightweight library implementing a compiler front-end capable of parsing most of the ANSI C, with support for many extensions used by the GCC (Gnu Compiler Collection) C compiler. Linus also implemented a simple client program called "check" that serves as a back-end and uses the library to tokenize, preprocess and parse a C file and print out warnings issued by the library. Additionally, it also makes few checks of its own. Due to the name "check" being too generic, Linus later decided to rename it to "Sparse". Linux kernel community started to use the Sparse tool for static analysis of the kernel code, especially since Sparse was able to notice mixing of user-space and kernel-space pointers, which is strictly related to kernel programming. Sparse is, however, also capable of more generic checks, like stricter type checking and pairing of synchronization functions [2, 4, 11].

The Sparse tool itself is one of many back-ends that use the Sparse library. These also include other tools for various purposes: visualization of C code by a graph, transformations into XML, etc. It is also possible to add a code generating back-end, which would result in a full-fledged compiler. In the following sections, we will take a more detailed look at how to use the Sparse tool to check the source code for errors and how the library can be used to construct a back-end.

Sparse can be downloaded from its homepage[1]. The Sparse project uses Git as its version control system. The latest development version of Sparse can be obtained using Git:

```
$ git clone git://git.kernel.org/pub/scm/devel/sparse/sparse.git
```

2.1 Using the Sparse Checker

This section will show what checks can be performed by the Sparse Checker. Whenever we say "Sparse" in this section, we are referring to the back-end, unless stated otherwise.

2.1.1 Invocation

Sparse is invoked with a list of filenames and options to turn on/off various warnings. A warning can be turned on by passing a command line option in the form `-W`*`warning_name`*. To disable a warning, an option in the form `-Wno-`*`warning_name`* must be passed. We can

[1] http://www.kernel.org/pub/software/devel/sparse/

`-D`*name*`[=`*value*`]`	Define *name* as a macro expanding to *value* (defaults to 1)
`-E`	Stop after the preprocessing phase
`-I`*dir*	Add *dir* to the list of directories Sparse will search for header files
`-include` *file*	Include *file* in the beginning of every processed file

Table 2.1: Some gcc-like Sparse options

also use the `-Wall` option to turn on all the warnings, except for those that are explicitly silenced by a `-Wno` option. For a complete list of warning options, see [12].

To better integrate with the Linux kernel build environment, which uses gcc, Sparse by default ignores command line options it does not know. Furthermore, it supports some gcc options that affect the preprocessing phase of compilation. Some of these are listed in Table 2.1.

Example

If we wanted to check a file named `foo.c` for all warnings, except for warnings about non-static variables and functions that have no previous declaration, we would execute Sparse as follows:

```
$ sparse -Wall -Wno-decl foo.c
```

If we also wanted to make sure that a C preprocessor directive `#include` looks for files in a non-standard directory, say, `/home/foo/include`, and define a macro `HAVE_CONFIG`, we would invoke Sparse like this:

```
$ sparse -Wall -Wno-decl -I/home/foo/include -DHAVE_CONFIG foo.c
```

2.1.2 Verifying Kernel Sources

Since the primary intentions behind creating Sparse were to find bugs in the kernel source code[2], it is very well integrated into the kernel build environment.

Linux is mostly built using GNU Make and Sparse is directly incorporated into its Makefile structure. Sparse checks the files at the build time. To build the Linux kernel, one would normally configure it using `make xconfig` and then executing `make`. However, if we also wanted to use Sparse to check files along the way, we would turn on the checking by adding `C=`*level* as an argument to `make`, where *level* is either 1 or 2. In case it is 1, `make` will only execute Sparse for files that are about to be compiled. If the *level* is set to 2, `make` will check every file, even if it already was compiled before. We can also tell `make` what binary it should use for checking, by adding `CHECK=`*/path/to/binary* to the command line. To pass additional flags to Sparse, add `CF=`*sparse_flags*.

When building the kernel with checking turned on, `make` will run the compiler on each file with appropriate command line arguments, which can be unique for each file. After the (successful) compilation, the same flags are passed to the checker. Hence, Sparse must be able to either use the passed flags or ignore them. Some of the preprocessor flags supported by Sparse are shown in Table 2.1. Without these flags, Sparse would fail to correctly parse the input files due to missing header files.

[2] Available at http://www.kernel.org/

Example

To compile and check the whole kernel tree with our own Sparse binary without warnings about mixing pointers with different address spaces, we could invoke Sparse as follows:

```
$ make C=2 CHECK=/home/foo/bin/sparse CF="-Wno-address-space"
```

2.1.3 Code Annotations

In order to use the more advanced tests Sparse has to offer, it is required that the tested source code contains Sparse-specific annotations. For this, Sparse uses the GNU C specific *attribute specifiers*.

An attribute specifier allows a developer to attach characteristics to declarations to allow the compiler to perform optimizations and better error checking [5]. It is used in the form `__attribute__((attribute-list))`. The *attribute-list* is a list of attributes separated by commas, and can possibly be empty. An attribute is an identifier optionally followed by a sequence of arguments enclosed in a pair of parentheses. Multiple attribute specifiers can appear as part of a declaration, adding the characteristics to the declarator [8, p. 299–302].

For example, in code Listing 2.1, we are declaring variable `foo` and specifying that gcc should not emit a warning in case it will not be used. Function `bar` is declared as non-returning.

```
1 int __attribute__((unused)) foo;
2 void bar(char *msg) __attribute__((noreturn));
```

Listing 2.1: Example use of `__attribute__`

Sparse makes use of this syntax and defines a few of its own attributes. A problem with this approach is that not all compilers support the `__attribute__` extension and even gcc does not know about attributes used by Sparse. This means that gcc, or any other compiler, would not be able to compile such a code. To solve this problem, Sparse automatically defines macro `__CHECKER__`, which can be used by the programmer to conditionally compile his code, as shown in code Listing 2.2.

```
1 #ifdef __CHECKER__
2 # define __user __attribute__((noderef, address_space(1)))
3 #else
4 # define __user
5 #endif
```

Listing 2.2: Conditionally use attributes if we are compiled by Sparse

Macro `__user` is now safe to be used with any compiler, but Sparse will recognize the attributes and will be able to use them. Most of the annotations used in the Linux kernel are conditionally defined in a similar manner. We will introduce more annotations as we move along. They are all taken from the kernel source code[3]

[3] `include/linux/compiler.h`

Address Space Checking

The `address_space` attribute is especially useful in the context of kernel programming, as there is a need to make sure that kernel functions will not accidentally access user space memory. Definition of a code annotation that uses address space attribute can be seen in code Listing 2.2. Address space is denoted by a number, which is by default 0. If we try to mix pointers of different address spaces as seen in the code Listing 2.3, Sparse will emit a warning on line number 3.

```
1 char *foo;
2 char __user *bar;
3 foo = bar;
```

Listing 2.3: Erroneous mixing of address spaces

Protection Against Dereference

The reader might have also noticed the `noderef` attribute in the definition of `__user`. An example of a bad dereference that Sparse will complain about can be seen in the code Listing 2.4. Both cases (line number 3 and 4) will be caught by Sparse. Note, however, that this is not the same as pointer mixing in code Listing 2.3, since we wanted to access the value pointed to by `bar`.

```
1 char foo;
2 char __user *bar;
3 foo = *bar;
4 foo = bar[0];
```

Listing 2.4: Forbidden dereference of a pointer declared with `noderef` attribute

Stronger Type Checking

C is relatively weakly typed, which can be source of many programming errors. Kernel code uses annotations (as seen in code listing 2.5) to enable stronger type checking using Sparse.

```
1 #ifdef __CHECKER__
2 # define __nocast   __attribute__((nocast))
3 # define __bitwise  __attribute__((bitwise))
4 # define __force    __attribute__((force))
5 #else
6 # define __nocast
7 # define __bitwise
8 # define __force
9 #endif
```

Listing 2.5: Annotations for stronger type checking

Using the `nocast` attribute in the type declaration will make sure that an implicit conversion to another type will be caught by Sparse. In this case, warnings from Sparse can easily be silenced by an explicit cast into the target type.

One typical coding error in the kernel code could be that the programmer would pass arguments to a function in a wrong order. If an allocation function takes a size argument and a set of flags, swapping these could lead to hard-to-find bugs [3]. Due to weak typing, a compiler will implicitly cast the value to the appropriate type and will not treat this as a bug.

Let us consider this set of bit flags and function `alloc`:

```
#define FLAG_A 0x1U
#define FLAG_B 0x2U

void *alloc(size_t size, unsigned int flags);
```

A programmer intending to allocate 128 bytes with both bit flags turned on could accidentally invoke the function like this:

```
void *ptr = alloc(FLAG_A | FLAG_B, 128);
```

This is obviously wrong, but the compiler will not complain. We can instead declare `alloc` like this:

```
void *alloc(size_t size, unsigned int __nocast flags);
```

Now, passing `128` as a second argument will trigger a warning, because a signed value needs to be implicitly cast into an unsigned value.

Even though this is very useful, it would be much more safer to use the `bitwise` attribute instead. Usage of a type with this attribute is heavily restricted by Sparse. It will only allow bitwise operations and any cast will need to use the `force` attribute. We demonstrate its usage in code Listing 2.6.

```
1  typedef unsigned int __bitwise flags_t;
2  #define FLAG_A ((__force flags_t)0x1)
3  #define FLAG_B ((__force flags_t)0x2)
4  void *alloc(size_t size, flags_t flags);
5
6  /* Bad */
7  alloc(128, 0x2);
8  alloc(128, (flags_t)0x2);
9  alloc(128, FLAG_A + FLAG_B);
10
11 /* Good */
12 alloc(128, 0);
13 alloc(128, FLAG_B);
14 alloc(128, FLAG_A | FLAG_B);
```

Listing 2.6: Usage of the `bitwise` attribute for stronger type enforcement

First, we will use `typedef` to create a new type, and add `__bitwise` annotation (line 1). We will then define a set of flags by force-casting integer values to `flags_t` (lines 2–3). It will be very hard to unintentionally misuse the second argument of `alloc`. Without using

the `force` attribute again, an acceptable argument can only be 0 (which is special) or a result of bitwise operations on flags defined by us (`FLAG_A` and `FLAG_B`).

Context Checking

Another very useful thing that Sparse can do is the tracking of static "code context" information [10]. A context has a name and a value, which can be changed as Sparse analyses the generated control flow graph. We can then put constraints on a function and specify values that the context must contain on entry and exit. If Sparse determines that there is a possible code path that would cause the context to be unbalanced, it will emit a warning. This of course is easily fooled, but should be sufficient in most cases [10]. The most useful use of these capabilities is pairing functions used for synchronization, e.g. making sure that if we call `lock()`, we don't forget to call `unlock()`.

To change the value of a context, we will use the `__context__(`*`name, value`*`)` statement, which will increment the value of context *name* by *value*. To put a constraint on a function, we will add attribute `context(`*`name,in,out`*`)` specifying that context *name* must have the entering value *in* and exiting value *out*. Annotations for use by the synchronization functions, as used in the Linux kernel, are shown in code Listing 2.7.

```
1  #ifdef __CHECKER__
2  # define __acquires(x) __attribute__((context(x,0,1)))
3  # define __releases(x) __attribute__((context(x,1,0)))
4  # define __acquire(x)  __context__(x,1)
5  # define __release(x)  __context__(x,-1)
6  #else
7  # define __acquires(x)
8  # define __releases(x)
9  # define __acquire(x) (void)0
10 # define __release(x) (void)0
11 #endif
```

Listing 2.7: Annotations for context validation

If the functions we want to annotate are called `do_lock` and `do_unlock`, we can define macros `lock` and `unlock` like this:

```
#define lock(x)   do { __acquire(x); do_lock(x); } while (0)
#define unlock(x) do { __release(x); do_unlock(x); } while (0)
```

Notice that in both cases we enclosed two statements in a `do { } while (0)` loop. This is a common C programming idiom to ensure that the resulting macro is safe to be used in one-line cycles that don't use braces. When Sparse is doing data-flow analysis, it will use the `__context__` statement to raise the context value. At the end of a function where this is used, the context must be zero, unless the function itself is marked with the `context` attribute.

Let us consider a C data structure named `object`. Every time we need to manipulate this object, we are required to call function `start_operation` that will initialize it and acquire a lock. Then, after we are done with the object, we need to call the `end_operation` function to clean it up and release the lock. Code for these functions can be seen in code Listing 2.8.

```
1  struct object {
2    lock_t lock;
3    /* Other members */
4  };
5
6  static void start_operation(struct object *obj)
7      __acquires(obj->lock)
8  {
9    lock(obj->lock);
10 }
11 static void end_operation(struct object *obj)
12     __releases(obj->lock)
13 {
14   unlock(obj->lock);
15 }
```

Listing 2.8: Annotation of functions acquiring and releasing a lock

With these annotations, Sparse will make sure that every use of `start_operation()` will be paired with exactly one `stop_operation()`, otherwise it will emit a warning.

2.1.4 Built-in Functions

Sparse provides two extra built-in functions that are especially useful to help avoid dangerous usage of C macros. If a C macro does not use its argument exactly once, it is dangerous since passed expression with side effects will be evaluated more than once (or not at all). Sparse can be used to protect from these kinds of errors by providing a predicate function `__builtin_safe_p(`*`expr`*`)` which will evaluate to 0 at compile time if *expr* might cause a side effect, and to 1 otherwise. An unsafe expression can, for example, be `a++`, `fun()`, etc.

To warn the programmer, we can use the `__builtin_warning([`*`cond`*`,] `*`message`*`)` function, where `cond` is an optional value determinable at compile time and *message* is a string. If *cond* is missing or evaluates to non-zero, *message* is printed by Sparse. An example C macro protected against unsafe usage is in the code listing 2.9.

```
1 #define MACRO(a) do { \
2   __builtin_warning(!__builtin_safe_p(a), \
3                     "Macro used in an unsafe way"); \
4   fun1(a); \
5   fun2(a); \
6 } while (0)
```

Listing 2.9: Using `__builtin_safe_p` to make a C macro safer

2.2 The Sparse Library

This section will talk about using the Sparse compiler front-end to construct various tools and back-ends. Whenever we say "Sparse" within this section, we are referring to the library, unless stated otherwise.

The library can be used for various purposes and will be useful to anyone who needs a C compiler front-end for his application. List of possible uses of Sparse include building a C compiler, or an analysis tool that searches for coding faults.

It is designed to be fast and easy to use, as demonstrated by the code Listing 2.10, taken from [9]. This code will initialize Sparse by arguments that were passed to it on the command line, see Section 2.1.1 for the list of possible options. It will then call function `action()` passing it all global symbols that were found. Of these, we will of course mostly be interested in functions.

An interesting fact is that as of this writing, the Sparse Checker itself is only 300 lines long, while the library is, in contrast, almost hundred times larger. This is brought on by the fact that the library does all the work, while Sparse itself only implements the context checks (described in 2.1.3). This means that every back-end can take the advantage of checks provided by Sparse.

```
1  #include <sparse/lib.h>
2
3  void action (struct symbol_list *syms)
4  {
5    /* Analyze symbol list */
6  }
7
8  int main(int argc, char *argv[])
9  {
10   struct string_list *file_list = NULL;
11   char *file;
12
13   action(sparse_initialize(argc, argv, file_list));
14   FOR_EACH_PTR_NOTAG(file_list, file) {
15     action(sparse(file));
16   } END_FOR_EACH_PTR_NOTAG(file);
17   return 0;
18 }
```

Listing 2.10: Basic usage of Sparse

In this section, we will introduce various functions and data structures used by Sparse and the client application for manipulation of the generated syntax tree and linearized byte-code. Among these functions are some that are general enough to be used outside of Sparse. First, we will take look at pointer lists. After that, we will quickly introduce functions used by both Sparse and the client program to issue warning and error messages to the user. After that, we will move on to discussing the structure of the syntax tree and linearized byte-code. These two sections are most important for us, as they represent the final output that Sparse produces for us.

Although we are trying to present as much information as possible, reader trying to write his own back-end will probably not be able to do it without reading at least some of the Sparse code. Our goal is to merely help the reader to quickly understand and learn the internals of Sparse. The fact that Sparse is an open-source project with active developers means that it can quickly change and this paper might potentially become obsolete. The

developers of Sparse do not even use data-encapsulation and do not maintain a stable Application Programming Interface.

We should also note that when discussing data structures, we often omit some of their members. These members are often only of interest to internal Sparse functions that make optimizations to the linearized byte-code, or are unimportant to us for other reasons. However, an interested reader is encouraged to read the source code. Written by skilled kernel programmers, Sparse can also serve as a valuable source of inspiration for beginning compiler writers and programmers.

2.2.1 Pointer Lists

Sparse uses singly-linked lists of pointers and provides some useful functions to manipulate them. Sparse functions often return multiple pointers in lists, which the caller then must use, so a basic grasp of these functions is necessary. A list data structure for each pointer type must be created separately, which helps to make the usage of lists more type-safer. However, because of this, basic functions such as `head` and `tail` must be separately created by the programmer himself (unless he wants to use a lot of explicit casts later).

Creating a List

Before using a pointer list, we first have to declare it with the `DECLARE_PTR_LIST(`*`list_name`*`,` *`type`*`)` macro, where *list_name* is the name of the list and *type* is the type of the pointers stored in the list. For example, in the code listing 2.11 we created a new list type with pointers to `struct foo`.

```
1 struct foo { int a; };
2 DECLARE_PTR_LIST(foo_list, struct foo);
```

Listing 2.11: Declaration of a new pointer list

A new empty list is created by defining a pointer to `struct foo_list` and initializing it to `NULL`. A pointer *ptr* can then be added to a list using the `add_ptr_list(`*`list_ptr`*`,` *`ptr`*`)` function, where *list_ptr* is a pointer to a list. Since a list is a pointer itself, this is really a double pointer.

A basic usage of a list can be seen in the code Listing 2.12. On line 1, we create a new `foo_list` named "list". To create a new `foo` data structure, we are using function `new_foo(`*`val`*`)` which will dynamically allocate it and initializes its member `a` to *val*. We use the `new_foo` functions to create two new `foo` data structures and add them to the list (lines 3–4).

```
1 struct foo_list *list = NULL;
2
3 add_ptr_list(&list, new_foo(1));
4 add_ptr_list(&list, new_foo(2));
```

Listing 2.12: Adding pointers to a list

Note that there is a convention to name lists in the *`datatype`*`_list` format, where *datatype* is the name of the data structures that we are holding. We will introduce these

lists without any further explanation, the reader is expected to be aware of this convention.

Traversing a list

The most often used operation on a list is traversal. A list can be traversed either from its first element to the last one or in reverse. For this, Sparse provides a pair of preprocessor macros. An opening macro will take a list and a pointer as an argument. The pointer must be of the same type as pointers in the list. The opening macro is then followed by a compound statement (enclosed in curly brackets) which contains code executed in each iteration. The pointer that we passed at the beginning will now point to the current element. After the compound statement, we must insert a closing macro, which has the same name as the opening one, but is prefixed by `END_` and takes the pointer we were using in the compound statement as the only argument.

To traverse the list from the first element to the last, we can use the `FOR_EACH_PTR` macro, pairing it with `FOR_EACH_PTR_END` macro. To traverse the list from the last element to the first, use `FOR_EACH_PTR_REVERSE` instead and end with `FOR_EACH_PTR_REVERSE_END`. An example is shown in code Listing 2.13. If we consider the previous code in Listing 2.12, the piece of code in Listing 2.13 will print out numbers 1 and 2.

```
1 struct foo *ptr;
2 FOR_EACH_PTR(list, ptr) {
3     printf("%d\n", ptr->a);
4 } END_FOR_EACH_PTR(ptr);
```

Listing 2.13: Traversing a list

Note that while the `continue` statement will work as excepted, the `break` statement will not (it will in fact behave in the same way as `continue`). The desired functionality, however, can be achieved through the use of `goto`.

Other List Operations

Other operations that we often need is to find out if a list is empty and get its first and last element. A macro that will tell us if *list* is empty is `ptr_list_empty(`*`list`*`)`. This is a very simple operation, since an empty list is one that equals to `NULL`, hence the `ptr_list_empty(`*`list`*`)` macro simply expands into `(`*`list`*` == NULL)`. Nevertheless, the code is more readable and safer for future changes if we use it.

To get the first element from a *list*, we can use the `first_ptr_list(`*`list`*`)` function. The problem, however, is that this function takes a `struct ptr_list *` argument and returns `void *`. This requires us to cast our list into `struct ptr_list *` and then cast the returned void pointer into the pointer type that we want. That is very tedious and doing it every time we need to do this operation is not comfortable, and can even lead to programming errors. The best practice for this case is to write a simple in-line wrapper function around `first_ptr_list()`. Since it will be in-lined, compiler will optimize it in such a way that it will incur no additional performance cost. This approach is also more type-safe, since the function will always require the list it was made for and the compiler will warn the programmer if he tries to use the function with an incorrect list.

To get the last element from a *list*, we can use function `last_ptr_list(`*`list`*`)` which uses the exact same syntax as `first_ptr_list`. We can see an example usage of these three

functions in code Listing 2.14.

```
1  static inline struct foo *first_foo(struct foo_list *head)
2  {
3    return first_ptr_list((struct ptr_list *)head);
4  }
5  static inline struct foo *last_foo(struct foo_list *head)
6  {
7    return last_ptr_list((struct ptr_list *)head);
8  }
9
10 if (!ptr_list_empty(list)) {
11   printf("%d\n", first_foo(list));
12   printf("%d\n", last_foo(list));
13 }
```

Listing 2.14: Miscelanous pointer list functions

There are some other functions and macros in Sparse for list manipulations that are useful, but less often used. For example, it is possible to modify the pointer while traversing the list, or concatenate and split lists. We do not discuss all of the list manipulation functions provided by Sparse here, but a curious reader can take a look at `sparse/ptrlist.h`.

2.2.2 Initialization of the Parser

Now that we have introduced the basic functions for manipulation of pointer lists, we can move on to describe functions that are used to initialize Sparse and generate a syntax tree.

Function `sparse_initialize()`

The first function that should be called by a client program is `sparse_initialize()`. The prototype of this function is in the code Listing 2.15. The `argc` and `argv` arguments have the same semantics as in function `main`. Thus, `argv` is an array of strings while `argc` the number of `argv` elements. The third argument is used to pass a pointer to an empty string list. Declaration of `string_list`, which is sometimes used in Sparse, is also shown in the code listing. Notice that this is an exception from the list-naming convention. The function returns a list of symbols that can be used to access the generated syntax tree. We will talk about them later in Section 2.2.4.

```
1 DECLARE_PTR(string_list, char);
2
3 struct symbol_list *
4 sparse_initialize(int argc, char **argv,
5                    struct string_list **filelist);
```

Listing 2.15: Prototype of `sparse_initialize()`

Using this function, the back-end developer will save himself from the burden of handling command line arguments. This is important, because we need to be able to handle some

pre-processing flags and correctly ignore some gcc flags, to properly integrate with the build environment, as described in Section 2.1.1.

The function also creates and/or initializes few important objects:

- identifier hash table,
- built-in symbols,
- built-in macro definitions,
- built-in functions.

It also declares the `__CHECKER__` macro which is important for the use of code annotations, as described in Section 2.1.3. It will then tokenize and parse files included with the `-include` option. The returned symbol list will global symbols found in these files.

Function cosparse()

After Sparse is initialized and we have a list of files, we can feed them to `sparse()`. The contents of these files will be parsed and a syntax tree will be generated. The list of global symbols found will be returned, as with `sparse_initialize()`. The prototype of sparse() follows:

```
struct symbol_list *sparse(char *filename);
```

The `sparse()` function must be called after `sparse_initialize()`, otherwise it will not work properly.

2.2.3 Producing Warnings and Errors

During the parsing stage and analysis of the source code, Sparse uses various functions to present warnings and errors to the user. Most of these warnings and errors directly relate to the source code that is being processed. The functions are also accessible to the programmer writing a back-end, since we might write our own checks that are trying to find bugs. For example, the context checking we described in Section 2.1.3 is implemented in the Sparse checking tool and not in the library itself, as is the case with most other checks.

Warnings would, however, not be very helpful if they would not identify the file name, line number and position of the offending construct. Since there are many object types in the syntax tree (symbol, statement, expression, etc.) that require this information, this data is concentrated in one data structure – `position`. All data structures used in the syntax tree have a `position` member, conventionally called `pos`. Members of the `position` data structure are listed in Table 2.2. The `stream` member points to an internal Sparse representation of input streams, which are identified by integers. A stream can either point to a file or standard input. We do not need to know much about Sparse streams, but one useful function is `stream_name`, prototyped in `sparse/token.h`:

```
const char *stream_name(int stream);
```

It will return the name of the stream, which is typically the name of the file. The returned pointer points to a static buffer, which means that it will be changed after another invocation of `stream_name()`. Hence, we should use it immediately and not keep it around. Because of these practices, Sparse is not very thread-safe, but is easier to work with as we do not have to worry to free allocated memory.

Type	Name	Description
`unsigned int:14`	`stream`	Stream number.
`unsigned int:31`	`line`	Line number.
`unsigned int:1`	`newline`	Set to 1 if this is the first token on the line.

Table 2.2: Members of the `position` data structure

Warning Functions

Both warning and error functions use a classic `printf`-like format string followed by a variable number of arguments. First argument is the `position` data structure. Prototypes of warning functions is here:

```
extern void info(struct position, const char *, ...);
extern void warning(struct position, const char *, ...);
```

An example message produced by the `warning` function:

```
example.c:10:7: warning: symbol 'a' was not declared. Should it be static?
```

Here we can see that the error occurred in file `example.c` on line number 10 and position 7. If we would use `info()` instead, the resulting message would be almost the same, except for the "warning: " prefix, which would be missing.

These kind of messages should point out dangerous places in the code that might be a source of errors. In reality, they might still be safe but could have signs of bad programming practise, like the warning we saw.

Error Functions

If the program encounters a more serious problem that might render the rest of the analysis unreliable, we need to issue an error and maybe even halt the program. For these kind of situation, we will use following functions:

```
extern void sparse_error(struct position, const char *, ...);
extern void error_die(struct position, const char *, ...);
void die(const char *, ...);
```

The first two functions are used in the same way as `warning()`, but the resulting message will say "error" instead of "warning". Additionally, `error_die()` will stop the program right after printing the error. Function `die()` can be used to exit after an error that was not caused by the incorrectness of the analysed source code. This could be an internal error indicating a bug in Sparse or the client application.

2.2.4 Syntax Tree

As already mentioned, `sparse()` will provide us with a list of globally defined symbols. To actually browse through the syntax tree and use it meaningfully, we need to know what objects it is composed of and how these objects are represented. This section will introduce syntax tree data structures and various functions that can be used to manipulate them.

Identifiers

A C *identifier* is defined as a sequence of letters, digits and the underscore character. It must not begin with a digit and must not have the same spelling as a keyword [6, p. 21]. An identifier can refer to various things, e.g.: functions, data structures, unions, variables, etc. [7, p. 195]. In the Sparse library, nn identifier is represented by `struct ident`, defined in `sparse/token.h`. Its members are described in Table 2.3.

Whenever the tokenizer encounters a token beginning with a letter or an underscore, it will look it up in the identifier hash table. If there is no identifier found, then a new one is created and added to the hash table. For this reason, Sparse uses the identifier structure to represent keywords as well. They are simply added to the hash table during the initialization stage and marked as special (fields `reserved` and `keyword`).

Type	Name	Description
`struct symbol *`	`symbols`	Pointer to symbol data structures that are referred to by this identifier. The symbols are linked by member `next_id` in `struct symbol`.
`unsigned char:1`	`reserved`	Set to 1 if the identifier is reserved.
`unsigned char:1`	`keyword`	Set to 1 if the identifier is a keyword.
`unsigned char`	`len`	Lenght of the `name` string.
`char[]`	`name`	Name of the identifier.

Table 2.3: Members of the `ident` data structure

A convenience function `show_ident()` can be used to get a textual representation of an identifier, mostly for debugging purposes. It is provided by `sparse/token.h`:

```
const char *show_ident(const struct ident *);
```

As with `stream_name()`, `show_ident()` will return a pointer to a static buffer, so it is necessary to be careful when using it.

Statements

The representation of a statement listed in the table quite straightforward in Sparse. Its members are listed in Table 2.4. Beside the two members, `statement` contains a union of several name-less structs, that are specific for each statement type (see `sparse/parse.h` for the complete list) Most of these are very simple.

Let us consider an if statement. It will typically consist of a condition and two statements. One statement is executed when the condition in runtime evaluates to a non-zero value. The other one is optional, and is executed if the condition does not evaluate to true. The corresponding Sparse `statement` will then be of type `STMT_IF`. The `if_conditional` member will point to the relevant `expression` data structure and members `if_true` and `if_false` will point to the two statements.

A *compound* statement (`STMT_COMPOUND`) is a statement composed of multiple statements enclosed in braces. It can be used anywhere where a statement can [6, p. 262]. The field `stmts` is a list of pointers to those statements. As a special case, the compound statement can be the body of a function. If that is the case, the `ret` field points to a symbol bound to the `return` indent. It is used by the linearization routines to serve as a target label for `return` statements inside of the block. Another special case is when the compound statement is created as a result of function inlining. Then, the `inline_fn` member will

Type	Name	Description
`enum statement_type`	`type`	Type of the statement (`sparse/parse.h` contains the list of all of them).
`struct position`	`pos`	Position of the statement.

Table 2.4: Members of the `statement` data structure

point to the symbol that represents the in-line function and the `args` member will point to a `STMT_DECLARATION` statement, which contains the list of symbols passed to the function. Such compound statement still uses the `ret` field for the `return` statement.

Iteration statements such as `for`, `while` and `do while` are all represented by one statement type, `STMT_ITERATOR`. This statement contains pointers to symbols representing `continue` and `break` jump targets (members `iterator_continue` and `iterator_break`) and a list of symbols that are declared in the cycle initial clause (member `iterator_syms`). This is only applicable to the `for` cycle. The `iterator_pre_statement` refers to the statement that is normally executed before the cycle begins. The `iterator_pre_condition` refers to the expression evaluated on each iteration,. If the pre-condition evaluates to a zero value, the cycle ends. The cycle body is represented by `iterator_statement`. The `iterator_post_statement` represents the statement executed by the end of each iteration. Finally, the `iterator_post_condition` refers to the cycle condition, evaluated at the end of each iteration. This condition determines if the cycle should continue. Not all statements and expressions are required. A missing expression will be treated as if it would always evaluate to true. This way we can express all three C iteration statements with one common statement data structure. Figure 2.1 shows these relations in a flowchart. Please note that a `continue` statement would jump to the *post-statement*, whereas the `break` statement would jump right in the end.

Expressions

The expression data structure, similarly to statements, contains some common data shared by all expressions and a union of different data structures that depend on the expression's type. The shared members are listed in Table 2.5. Expressions can be classified into several groups:

- *primary*: `EXPR_VALUE`, `EXPR_FVALUE`, `EXPR_STRING`, `EXPR_SYMBOL`
- *unary*: `EXPR_UNOP`, `EXPR_PREOP`, `EXPR_POSTOP`
- *type information*: `EXPR_CAST`, `EXPR_SIZEOF`, `EXPR_ALIGNOF`
- *binary*: `EXPR_BINOP`, `EXPR_COMMA`, `EXPR_COMPARE`, `EXPR_LOGICAL`, `EXPR_ASSIGNMENT`
- *ternary*: `EXPR_CONDITIONAL`, `EXPR_SELECT`
- *other*: `EXPR_CALL`, `EXPR_LABEL`

Expressions involving operators (unary, binary and `EXPR_CALL`) use the `op` field to hold the character that represents the operator. This means that expression $expr_1$ `+` $expr_2$ will be represented by an `expression` data structure with type `EXPR_BINOP`, and op `'+'`.

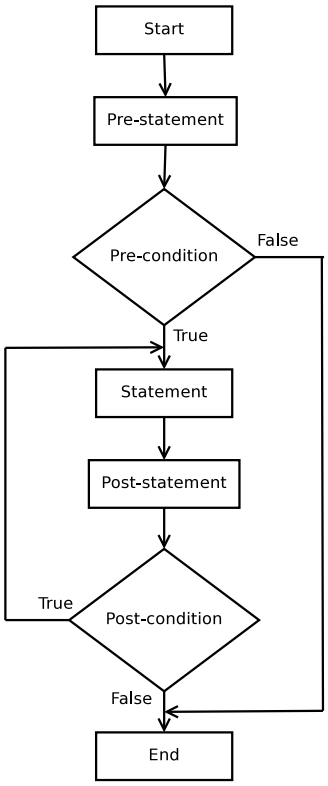


Figure 2.1: Diagram visualizing the iterator statement

Furthermore, its members `left` and `right` will point to the representation of $expr_1$ and $expr_2$ respectively.

The `op` member contains the operator character, when the expression involves an operation (if it is one of unary, binary or `EXPR_CALL`). If the expression type is `EXPR_BINOP` (binary operation), a valid value could be `'+'`, `'*'`, or some other C character literal, for one-character operators. Some C operators however are longer than one character. To accommodate for this, `sparse/token.h` defines `special_token` enumeration constants which can be used. All `special_token` constants are larger than 255 to avoid clashes with one-character operators. For example, the `<=` (less or equal) will be represented by `SPECIAL_LTE`. Some even take up three characters, like `>>=` (right shift assignment) which is represented by `SPECIAL_SHR_ASSIGN`.

Type	Name	Description
`enum expression_type`	`type`	Expression type (see `sparse/expression.h`).
`int`	`op`	Operator (see `sparse/token.h` for all operators).
`struct position`	`pos`	Position of the expression.
`struct symbol *`	`ctype`	Type of the expression.

Table 2.5: Members of the `expression` data structure

Sparse expression parser also makes some transformations on expressions that are interchangeable. Array subscription will be transformed into addition and dereference. Indirect member selection operation will be transformed into dereference and direct member selection operation. See Table 2.6.

From	To
$expr_1$`[`$expr_2$`]`	`*(`$expr_1$ `+` $expr_2$`)`
$expr_1$`->member`	`(*`$expr_1$`).member`

Table 2.6: Equivalent transformations done by the expression parser

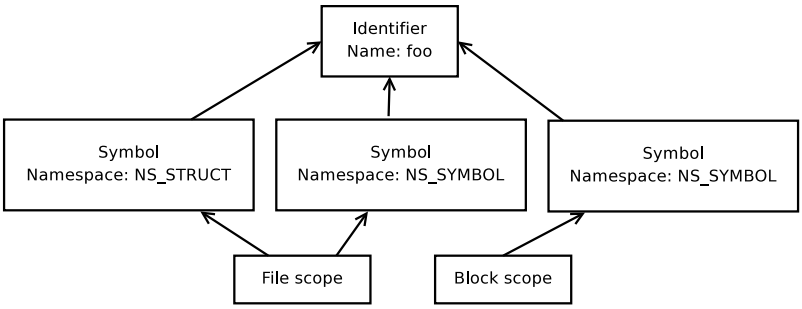


Figure 2.2: Relation between identifier, symbols and scopes

The `ctype` field is filled in by constant expression evaluation functions. The expression parser only fills this in for integer literals. After the parsing is done, `sparse()` invokes the `evaluate_symbol()` function:

```
static struct symbol *evaluate_symbol(struct symbol *sym);
```

The syntax tree is traversed and for every expression that is found, its `ctype` member is filled in. For every implied cast there is a new expression created, with different type.

Symbols

Symbols can represent types, variables, functions, labels, and some other things. A C identifier represented in Sparse is only a string with no semantics. A Sparse Symbol, on the other hand, is defined by a name and type information.

In C, the same identifier can be used to name different things. This is called *overloading*, and the correct resolution is made by looking at the context where the identifier is used [6, p. 77]. Symbol then belongs into, what is called an *overloading class*, or *name space* [6, p. 78]. In Sparse, the term "name space" is preferred. There can also be another symbol with same name space, but with different *scope*. In code Listing 2.16, we can see that identifier `foo` is used three times and each one is a different symbol. The situation is also illustrated in Figure 2.2.

```
1 struct foo foo;
2
3 void fun(void)
4 {
5   int foo;
6 }
```

Listing 2.16: Symbol `foo` with different name space and scope

To enable correct overloading, the symbol data structure contains information about its name space and scope, as can be seen in Table 2.7.

The symbol data structure contains fields used by all symbols and a union of three data structures. Using these data structures is then only allowed if the symbol belongs to the right name space. Two of these are used for preprocessor symbols (directives like `#define` and macros). However, the directives belong to the file scope, and hence are not included in the syntax tree generated by `sparse()`. Also, preprocessor macros are expanded during the preprocessing stage. The third data structure applies for the other name spaces has its members listed in Table 2.8. Since we don't have to worry about preprocessor symbols and macros, we can treat symbols as if these members were shared.

Type	Name	Description
`enum type`	`type`	Type of the symbol (see `sparse/symbol.h` for list of types)
`enum namespace`	`namespace`	Name space this symbol belongs to (see `sparse/symbol.h` for list of name spaces).
`struct position`	`pos`	Where this symbol was declared.
`struct ident *`	`ident`	Identifier with which this symbol is associated.
`struct symbol *`	`next_id`	Next symbol that shares the same identifier. The symbol is unlinked when the scope terminates.
`struct scope *`	`scope`	Scope of this symbol. Only last until the scope is ended.
`void *`	`aux`	Pointer for auxiliary data, to be used by the back-end.

Table 2.7: Members of the `symbol` data structure

Type	Name	Description
`unsigned long`	`offset`	Offset from the beginning of a data structure. Only applicable if this symbol is a member of a data structure.
`int`	`bit_size`	Size in bits, -1 if not applicable.
`struct expression *`	`array_size`	Expression used to specify array size. Applicable if the symbol type is `SYM_ARRAY`.
`struct ctype`	`ctype`	Type of the symbol.
`struct symbol_list *`	`arguments`	List of function arguments. Applicable if the symbol type is `SYM_FN`.
`struct statement *`	`stmt`	`case` statement.
`struct symbol_list *`	`symbol_list`	List of data structure members. Applicable if the symbol type is `SYM_STRUCT` or `SYM_UNION`.
`struct expression *`	`initializer`	Initializer expression for declarations, `enum` members, or expression used in `typeof()`.
`struct entrypoint *`	`ep`	Entry point to the linearized byte-code. See Section 2.2.5.

Table 2.8: Additional members of the `symbol` data structure

Representation of a C Type

A C type is represented by the `ctype` data structure, assisted by the `symbol` data structure. The list of its members can be seen in Table 2.9. The `base_type` member points to the symbol that represents this type. Symbols for basic built-in types are created by `init_ctype()`, in file `sparse/symbol.c`. This function is in turn called by `sparse_initialize()`. The `ctype` structure itself then only helps by adding modifiers to the type, since having a symbol for every modification would be impractical.

The `modifiers` field is set to a value obtained by doing a bitwise OR with multiple modifier constants. The constants can be classified as follows:

- *storage class specifiers*: `MOD_AUTO`, `MOD_REGISTER`, `MOD_STATIC`, `MOD_EXTERN`,
- *type qualifiers*: `MOD_CONST`, `MOD_VOLATILE`,
- *type specifiers*: `MOD_CHAR`, `MOD_SHORT`, `MOD_LONG`, `MOD_LONGLONG`,
- *function specifier*: `MOD_INLINE`.

Type	Name	Description
`unsigned long`	`modifiers`	Modifiers. List of them is in `sparse/symbol.h`.
`unsigned long`	`alignment`	Alignment in bytes.
`struct context_list *`	`contexts`	List of contexts (applicable to functions).
`unsigned int`	`as`	Address space, default 0.
`struct symbol *`	`base_type`	In case this is not a basic type (e.g. we're a struct) point to the symbol that represents the type.

Table 2.9: Members of the `ctype` data structure

Scope

Scope of a declaration is an area of a C program in which the declaration is visible [6, p. 75]. The `symbol` data structure contains a `scope` field that temporarily points to the scope of that symbol. The structure representing the scope in Sparse is defined in `sparse/scope.h`. List of its members can be seen in 2.10.

Type	Name	Description
`struct token *`	`token`	Starting token of the scope.
`struct symbol_list *`	`symbols`	List of all symbols in this scope.
`struct scope *`	`next`	Next scope.

Table 2.10: Members of the `scope` data structure

Symbol scoping in Sparse is relatively simple. There are four global pointers to scope structures: `block_scope`, `function_scope`, `file_scope` and `global_scope`. Each represents a different scope type. With the use of the `next` field, the scopes are stacked on top of each other, and the four pointers serve as starting points. There is always only one

global and file scope. The file scope is destroyed and created on each `sparse()` invocation. Function and block scopes are restarted as well when this happens. Starting and ending of a function scope will also have the same effect on the block scope as well.

Normally, the scopes are created and destroyed by Sparse, but the global and file scopes can be used by the programmer. The file scope particularly can be of interest to someone who also wants to access preprocessing directives and macros, which use the file scope. An example usage of this can be seen in `c2xml.c` and `ctags.c` distributed with Sparse. The file scope will be destroyed and a new one will be created with each invocation of `sparse()`, so any analysis requiring macros has to be done between the calls.

2.2.5 Linearized Byte-code

Sparse also provides a set of functions and data structures for generation of "linearized byte-code", which is basically a Control Flow Graph. The byte-code is represented by entry points, basic blocks, instructions and so-called *pseudos*.

An entry point contains a list of basic blocks. A basic block consists of series of instructions with no branches, except for the last instruction, which is always a branch or a return. The branch will always target the beginning of a basic block. There is always only one basic block for each entry point that contains the return instruction [13].

A simple code example in Listing 2.17 when converted to the linearized byte-code will look as in Figure 2.3. We can see here that BB_1 is the first basic block. The `seteq` instruction will compare `%arg1` with a constant literal 1 and store the result into register `%r2`. It will then branch off (instruction `br`) depending on the result. If the equality holds, it will jump to BB_2, which will simply call function `f(1)` and unconditionally branch to BB_5. Otherwise, it will jump to BB_3 and do another test, and so on. In any case, the control flow will eventually get to the terminating basic block BB_5. After call `f(3)` is made, the return instruction will be executed. One could wonder why the registers are numbered 2 and 4 instead of 1 and 2. This is because Sparse uses various optimization techniques which can optimize whole basic blocks out. This can cause registers to get lost and Sparse does not make any effort to reuse them. It should be noted that this code does not really get executed (although a back-end could do that), the byte-code is only used for analysis.

```
1 void fun(int arg)
2 {
3   if (arg == 1)
4     f(1);
5   else if (arg == 2)
6     f(2);
7   f(3);
8 }
```

Listing 2.17: Simple code used to visualize the linearized byte-code

To linearize a symbol, one would pass it to the `linearize_symbol()` function:

```
struct entrypoint *linearize_symbol(struct symbol *sym);
```

The resulting `entrypoint` structure can then be used to access the basic blocks. If the symbol is not a function, `NULL` will be returned. We will now describe the data structures in a bottom-up manner, starting at pseudos and finishing up by entry points.

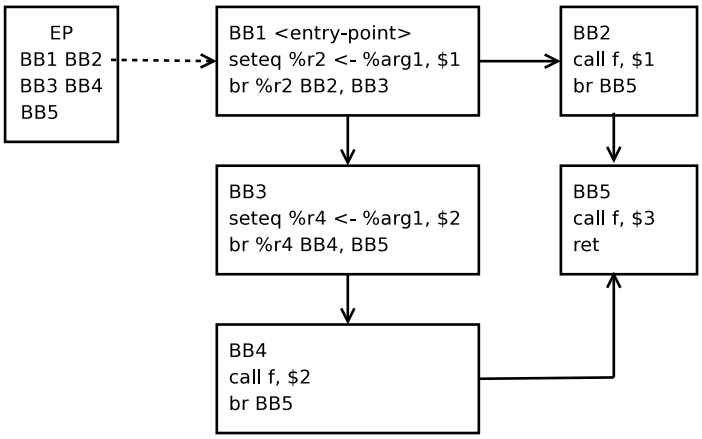


Figure 2.3: Entry point and basic blocks visualizing code 2.17

Pseudo

Pseudo represents a variable and is used as an argument for instructions. As with other Sparse data structures, `pseudo` contains members that are shared by all pseudo types and a union of members that depend on the type. The members are listed in Table 2.11, the list of union members is in the second part of the table. There is a typedef in `sparse/lib.h` that is important to know:

```
typedef struct pseudo *pseudo_t;
```

Type	Name	Description
`int`	`nr`	Pseudo number, unique for pseudos of the same type. Not applicable to pseudos of type `PSEUDO_VOID`, `PSEUDO_SYM` and `PSEUDO_VALUE`.
`enum pseudo_type`	`type`	Pseudo type.
`struct pseudo_user_list *`	`users`	List of pseudo users.
`struct ident *`	`ident`	Identifier of the symbol this pseudo was created from.
Depending on type		
`struct symbol *`	`sym`	Symbol this pseudo was created from. Only applicable to pseudos with type `PSEUDO_SYM`.
`struct instruction *`	`def`	Instruction where this pseudo is created. Applicable to pseudos with types `PSEUDO_REG`, `PSEUDO_ARG` and `PSEUDO_PHI`.
`long long`	`value`	Value of the pseudo. Only applicable to pseudos with type `PSEUDO_VAL`.

Table 2.11: Members of the `pseudo` data structure

A special global pseudo called `VOID` is used as a return value indicating an invalid pseudo, similarly as `NULL` is used by functions returning a pointer. Pseudos that are numbered have a number that is unique for their type across the entire run-time of the program. Pseudos of types different than `PSEUDO_VOID`, `PSEUDO_SYM` and `PSEUDO_VAL`

Here is the list of pseudo types used by Sparse:

- `PSEUDO_VOID`: used for the special `VOID` pseudo.
- `PSEUDO_REG`: register.
- `PSEUDO_SYM`: symbol.
- `PSEUDO_VAL`: value.
- `PSEUDO_ARG`: function argument.
- `PSEUDO_PHI`: special pseudo for ϕ functions.

Every instruction that uses a pseudo can be found using the `users` field, which is a list of `pseudo_user` data structures (Table 2.12). The pseudo user structure contains a pointer to the instruction that uses the pseudo and a pointer to `pseudo_t`. Since `pseudo_t` is a pointer by itself, this is a double pointer and allows us to easily replace the pseudo in the instruction itself. The code Listing 2.18 demonstrates this.

Type	Name	Description
`struct instruction *`	`insn`	Instruction using the pseudo.
`pseudo_t *`	`userp`	Pointer to the address where the pseudo pointer is stored in the instruction.

Table 2.12: Members of the `pseudo_user` data structure

```
1 pseudo_t old;
2 pseudo_t new;
3 struct pseudo_user *pu;
4
5 FOR_EACH_PTR(old->users, pu) {
6   *pu->userp = new;
7 } END_FOR_EACH_PTR(pu);
```

Listing 2.18: Replacing pseudo `old` for pseudo `new`

Instruction

Sparse uses a relatively high-level instruction set that often corresponds directly to the C operations [13]. Members of the `instruction` data structure are listed in Table 2.13. Most important one is the `opcode` field which specifies what kind of instruction this is. Additionally to the fields listed in Table 2.13, `instruction` contains one big union of data structures that are each used by different type of instruction. The whole data structure can be seen in `sparse/linearize.h`.

Most of the instructions take arguments, make a calculation and store it into the `target` pseudo. The jump instructions use the `cond` field instead of `target`. It is used to determine where to jump. The branch instruction (`OP_BR` has two possible basic blocks to jump to. They are pointed to by `bb_true` and `bb_false`. If one of them is `NULL`, the branch will act as an unconditional `goto`. An other case is the switch instruction (`OP_SWITCH`). It

Type	Name	Description
`unsigned int:8`	`opcode`	Instruction opcode, see cosparse/linearize.h for a complete list.
`unsigned int:24`	`size`	Size of instruction operands.
`struct basic_block *`	`bb`	Basic block this instruction belongs to.
`struct position`	`pos`	Position. Inherited from the symbol, statement or expression that was being linearized while the instruction was created.
`struct symbol *`	`type`	Type of the resulting pseudo.
Depending on opcode		
`pseudo_t`	`target`	Pseudo getting the result of the instruction.
`pseudo_t`	`cond`	Condition for branch and switch instructions (`OP_BR` and `OP_SWITCH`).

Table 2.13: Members of the `instruction` data structure

contains a list of the `multijmp` data structures, which contain a target and an integer range (Table 2.14). If `cond` is in range, the jump will be made. If `end` is less than `begin`, the target will be treated as default.

Type	Name	Description
`struct basic_block *`	`target`	Jump target.
`int`	`begin`	Minimum value.
`int`	`end`	Maximum value.

Table 2.14: Members of the `multijmp` data structure

Other important instructions are phi (`OP_PHI`) and phisrc (`OP_PHISOURCE`). They are used to implement ϕ-functions. The phisrc instruction simply takes one argument, `phi_src` and has target set to a `PSEUDO_PHI` pseudo. It also maintains a list of instructions that use this phi, `phi_users`. The phi instruction does the reverse thing – it takes a list of phis (field `phi_list`) and selects the appropriate one and puts it into the target pseudo. The selection is done by determining where we came from. Recall that the `PSEUDO_PHI` pseudos have pointers to the instruction they were defined in. An example code Listing 2.19 demonstrates how phi instructions are created. See Figure 2.4 to see the resulting linearized byte-code.

Basic Block

Basic block is an ordered list of instructions that contains one jump instruction at the end. It must not contain any other jump instruction. It is not possible to jump at any other instruction other than the first one in the basic block.

Basic blocks are useful for many optimization techniques, such as finding local common subexpressions or dead code elimination [1, p. 533–535]. Sparse implements both of these techniques as part of the code linearization.

The Sparse basic block data structure also contains some other fields used internally by Sparse optimizations routines. We will not be listing them here as they are not that important for us.

```
1  void f(int);
2  int g(void);
3
4  int fun(int arg)
5  {
6          int i;
7
8          if (arg)
9                  i = g();
10         else
11                 i = g();
12         f(i);
13 }
```

Listing 2.19: Code to demonstrate phi instructions

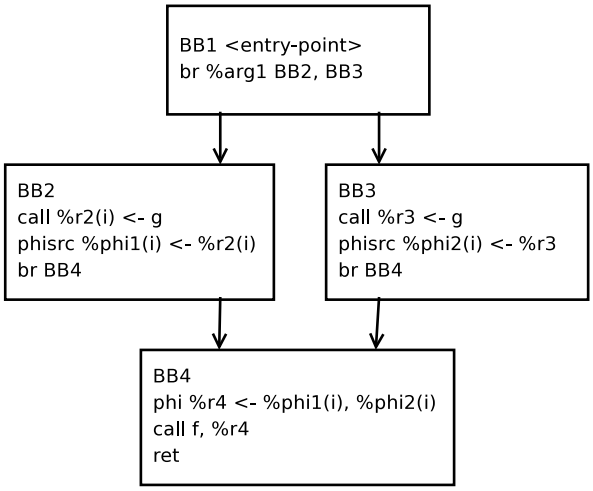


Figure 2.4: Basic blocks visualizing code 2.19

Further, a basic block contains a list of parents and children. A parent basic block contains a jump instruction that might jump to the child basic block.

Entry Point

The `entrypoint` structure represents an entry point to the linearized byte-code. List of members is in Table [13].

Type	Name	Description
`struct position`	`pos`	Position.
`struct entrypoint *`	`ep`	Entry point this basic block belongs to.
`struct basic_block_list *`	`parents`	List of basic blocks that jump to this basic block.
`struct basic_block_list *`	`children`	List of basic blocks that can be jumped to from this basic block.
`struct instruction_list *`	`insns`	List of instructions.

Table 2.15: Members of the `basic_block` data structure

Type	Name	Description
`struct symbol *`	`name`	Symbol of the linearized function.
`struct symbol_list *`	`syms`	List of symbols that are used in the code.
`struct pseudo_list *`	`accesses`	List of symbol pseudos that are accessed in the code.
`struct basic_block_list *`	`bbs`	List of all basic blocks in the CFG.
`struct instruction *`	`entry`	The first instruction of the graph. This is usually a special "entrypoint" instruction that contains the list of function arguments.

Table 2.16: Members of the `entrypoint` data structure

Chapter 3

Mygcc

Mygcc is a patched version of gcc, extended to perform user-defined checks by searching for patterns in C source code. It uses the *unparsed pattern matching* technique, a new approach able to recognize patterns in abstract syntax trees (AST), introduced in [15].

Mygcc was created by Nic Volanschi in 2004 [14]. The author describes his motivation for writing it in [16]. It was author's feeling that despite many recent advances in software checking tools, the usage of these tools in software projects is practically non-existent. As a solution, the author proposes to integrate the software checker with existing compilers to allow wide-spread adoption by programmers. This is a problem, since the usual methods are either hard to implement, or hard to use. The checks must also be very fast, or else compilation would take too long. To express these user-defined checks, Mygcc uses the *Condate* language.

The unparsed pattern matching method is easy to implement and use – the patch for gcc[1] is merely 3000 lines long. Most of the time, the use of these checks will not have a substantial performance overhead on the compilation time [16, p. 2]. After Mygcc, Nic Volanschi also created the myPatterns library[2] which provides the unparsed pattern matching capabilities applicable to any data structure used in a program (as opposed to just text).

Interestingly, even with this simple approach, Mygcc's practical capabilities are very impressive. In [16, § 6], the author uses results of a previous study involving detection of bugs in the Linux kernel to assess efficiency of Mygcc. The study was conducted on kernel version 2.4.1, finding and confirming more than 500 bugs using checkers written in the Metal language. After writing a set of checks in Condate that were supposed to mimic some of the Metal checkers used in the study and testing them on the same code base, the results were surprisingly good. Even though the Condate language does not have the expressive power of Metal, the results of checks for classes of problems addressable by Mygcc were almost the same.

By extending the compiler with capabilities to make simple user-defined checks, developers are able to find bugs early in the development process. The integration to an existing build infrastructure is even easier than with Sparse, which we discussed in Section 2.1.1, on page 3. Having the checking done in compiler also avoids duplication of compiler front-end code that does the parsing. Duplication of code almost always implies repetition of bugs already solved by others. Also, the programmer is more likely to choose to use simple code checking functionality in his compiler, than searching for a complex code analysis tool, not

[1] Available at http://gcc.gnu.org/ml/gcc-patches/2007-04/msg00822.html
[2] Homepage at http://mypatterns.free.fr/index.html

to mention the need to learn to use it.

In the following sections, we will introduce the concept of unparsed patterns and describe the Condate language that harnesses these patterns in order to allow the definition of queries that ask reachability questions about source code. We will then discuss limitations that are inherent to both the algorithm and the implementation. Finally, we will conclude this section by discussing the current state of the project and its future. Please note that the following text is not an original research in any way. It is merely a summary of the work done in [15, 16].

3.1 Unparsed Patterns

Introduced in [15], the *unparsed pattern matching*, as the name suggests, is a technique for matching syntax trees that does not require parsing of the regular expression. Instead, it unparses the abstract syntax tree, generating its string representation and tries to match it with the pattern. This technique is language-independent, but we will only use examples written in C.

Patterns also contain so-called *meta-variables*. A meta-variable matches any part of the syntax tree. Named meta-variables must match the same syntax tree either in the scope of the pattern (local) or across multiple patterns (global). An anonymous meta-variable can always match any syntax tree.

Recognition of meta-variables in patterns is the only parsing required and is therefore kept very simple. In patterns Mygcc uses, meta-variables are represented by letters, prefixed by the "%" character. An uppercase letter represents a global variable, while a lowercase letter represents a local variable. Anonymous meta-variables are represented by "%_".

Let us consider a simple pattern "%x = %x + 1". It will match any incrementation of an lvalue[3] %X, which matches any syntax tree. This means that it will also match an expression "*(a + 1) = *(a + 1) + 1", since the AST representations of "*(a + 1)" are compared here. Not only that, but Mygcc will also match this pattern to "*(a + 1) = 1 + *(1 + a)".

In order to use this approach, we must be able to "print" the syntax tree in a consistent manner. Thus, writing a printer for the syntax tree is required and can be the one obstacle in implementing this technique in an existing compiler. The source of problems can be the compiler's representation of the tree. As an example, the syntax tree representation used by Sparse could be considered too high-level, since an AST representation of "a = (`int`)a + 1" will not match the pattern we defined. The problem is that the syntax trees don't match, because casting "a" to `int` will yield a different syntax tree. The AST printer has to account for this and ignore casts, thus leaving us without an option to do any checks that involve casts.

However, the pattern does not match any incrementation of a variable, since we could also do that in C by writing "a++", which will not match our pattern. This also depends on how the syntax tree is built. If we used Sparse's linearized byte-code, there would not be a difference between "a = a + 1" and "a++". However, using Sparse's linearized byte-code is out of the question, since it optimizes the code into such an extent that recognizing other C constructs is not feasible. We will show a user-level solution to this problem in the next section. Mygcc uses GIMPLE[4], a tree representation in gcc.

[3] Because only lvalues can be on the left side of an assignment in C.

[4] http://gcc.gnu.org/wiki/GIMPLE

3.2 The Condate Language

The patterns are not expressive enough by themselves to be useful for control flow analysis. For example, they cannot be used to detect locking problems. One pattern by itself cannot even match an incrementation of a variable, as we saw in the previous section. To be able to find these problems, we need a framework that would allow us to not only match patterns, but would also allow us to find path from one pattern to another, optionally fulfilling a condition about what pattern must not be matched between them.

This is where the *Condate* language comes in. It was created for Mygcc to formalize *constrained reachability queries* using unmatched patterns. These queries are also called *condates*. The following grammar[5] is used to express condates:

$$S \rightarrow \textbf{from } D \textbf{ to } D \textbf{ avoid } D \tag{3.1}$$

$$D \rightarrow E \mid E \textbf{ or } E \tag{3.2}$$

Where S is the start symbol, or a "condate". D is the disjunctive pattern and E is an unparsed pattern. If at least one pattern E in a disjunctive pattern D matches the source code, then we say that D matches the source code. In order for condate "**from** D_1 **to** D_2 **avoid** D_3" to match, there must be a path from a match of D_1 to match of D_2. Additionally, D_3 must not have any matches on the path.

Thus, condates are used to represent control flow graphs (CFGs) of of code with defects. As an example, a memory leak could be caught by this condate: It will match any CFG in

```
1 from "%X=malloc(%_)" to "return" or "return %_" avoid "free(%X)"
```

Listing 3.1: Condate for finding memory leaks

which `malloc()` is called and the returned value is not passed to `free()` before returning from a function. We can also see the usefulness of global meta-variables. In the context of the Condate language, a global meta-variable is always visible throughout the condate. Another useful practice is the use of the disjunctive pattern to match two kinds of returns. This can be used as well to solve the incrementation problem mentioned in the previous section. To match incrementation correctly, we would express it as:

from "%X=%X+1" or "%X++" or "++%X"

Notice how we only use the "from" part to match pattern without any path. In reality, the simple "%x=%x+1" is enough to match even "a++". As already mentioned, all of this depends on the representation of the used AST. Simple patterns like this can be useful. For example, the following pattern will catch possible memory leak in case the `realloc()` function will fail and the old pointer gets lost:

from "%X=realloc(%X)"

Other useful and simple patterns could be used to detect the usage of deprecated function interfaces, like the dangerous `gets()`.

3.3 Usage

Since Mygcc is just an extended version of gcc, it is invoked in the same way. Mygcc provides two additional command line options for specification of condates. We can specify

[5] We only present a simplified condate grammar for brevity. See [16, § 3.2] for the complete grammar.

them directly on the command line or make Mygcc read them from a file. To specify a condate on the command line, we would invoke Mygcc with `-ftree-check=`*`condate`*, where *condate* is the condate string. We might need to quote the string to prevent it from being misinterpreted by the shell. If we have condates in a *file*, we would instead use the `-ftree-checks=`*`file`* option. When writing condate files, an additional syntax is used to enable naming of condates and associating warning messages with them. An example of this is shown in code Listing 3.2.

```
1 condate increment {
2   from "free(%X)"
3   to "*%X" or "%X->%_"
4   avoid "%X==%_"
5 } warning("Dereferencing released resources");
```

Listing 3.2: Condate to catch NULL dereference

3.4 Limitations

The design goals of Mygcc were not to make a complex static analysis tool, but instead, build a simple tool that is easy to use and integrated into the compiler. Two notable features that Mygcc lacks is alias information and semantic information[6].

3.4.1 No Semantic Information

Mygcc does not know about data types. The complete Condate grammar does have the power to use semantic constraints, but Mygcc does not implement them. The semantic constraints were meant to use internal gcc predicates. The lack of semantic constraints in the Mygcc implementation contrasts greatly with the Sparse checker. In Section 2.1.3, we used the `__user` annotation to mark pointers as belonging to user-space, thus making sure Sparse would warn if they either got dereferenced or mixed with pointers that use different address space. With the technique used by Sparse, it is almost impossible to do any of these mistakes unwillingly. If Mygcc would support semantic constraints, it would perhaps be able to easily express much of what Sparse can do.

One of the Metal checkers Mygcc tried to reproduce was able to make sure that no user pointers were dereferenced. Mygcc was only able to use a very limited approach. The condate used to detect dereference of user-space code in kernel first matched the `copy_from_user()` function. This function uses one parameter that is the user pointer and copies the user data to a kernel region of the memory. Mygcc matched the user pointer passed to the `copy_from_user()` with a global meta-variable and then tried to match an AST dereferencing the AST matched by the meta-variable. This solution is obviously far from ideal.

3.4.2 No Alias Information

Because Mygcc lacks any alias information, it can miss some bugs that involve pointer aliases. Considering the memory leak finding condate in code Listing 3.1, a false-positive

[6] Although semantic features are not implemented, they are used in [16, § 3.1.4]

would be raised for code in Listing 3.3. Since Mygcc is not aware that the code is in fact equivalent to calling `free()` on the `a` pointer, a warning is issued.

```
1 static void fun(void) {
2   struct foo *a, *b;
3   a = malloc(sizeof(struct foo));
4   b = a;
5   free(b);
6   return;
7 }
```

Listing 3.3: A false-positive for the memory leak test

During the comparison of Mygcc with checkers using Metal, two bugs went unnoticed because of this insufficiency [16, § 6.2.1]. However, these two were just a fraction of all the bugs that Mygcc was able to find. Considering how small Mygcc is, this seems like only a minor drawback. We can also conclude that these types of bugs are relatively rare. If this was not the case, there would be more than two defects of this class found in the Linux kernel by the Metal checkers.

3.5 Current Status and the Future

As of this writing, the Mygcc patch is not applied upstream. According to [14], it was rejected for inclusion in gcc because it uses "pretty-printer" to represent AST as a string. This for some reasons is not acceptable, as the pretty-printer is "possibly evolving" part of gcc.

For now, Mygcc is available as a separate branch, for anyone interested. Unfortunately, this is the exactly opposite effect of what the original intentions behind Mygcc were. Instead of being integrated into a widespread compiler, one needs to download a whole gcc branch and compile it to be able to try it out. However, the concept of unmatched pattern matching is still an interesting one. It would be interesting to, for example, use such an algorithm together with the Sparse library in order to provide a light-weight tool with the same power that Mygcc provides.

Most importantly, finding a way to incorporate Mygcc into the main gcc branch would be very beneficial to both open source communities and researchers working in the static code analysis field. If the checks would become standard gcc features, people would most likely embrace it fairly quickly. Another preferable outcome would be if the developers of software libraries started to provide a set of checks that the programmer using the library could use. Often, there are some rules that the programmer must abide to in order for the library to work correctly. The only way the developer of the library can "enforce" this rule is to document it. As programmers do not always read the documentation, this is an easy source of bugs that could be found and solved by a trivial static code analysis.

Chapter 4

Conclusion

During the study of Sparse and Mygcc, I conducted several experiments with these tools in order to assess their capabilities. I also found some bugs, either while reading the source code or when conducting the tests. The tests are located on the enclosed DVD, see appendix A. Besides the tests and experiments that are described here, I also attempted for a very brief time to combine Mygcc and Sparse together, which would help to Mygcc's current situation, as a non-merged patch. However, this attempt fell short fairy quickly. The porting of the code would require some knowledge of gcc's internal data structures, which is why I decided, to leave this endeavor for another time.

4.1 Experiments

4.1.1 Sparse

With the Sparse tool, I verified that Sparse is very good at strong type checking and guarding the pointers from mixing, and user pointers from dereferencing. In comparison with Mygcc, Sparse does this job much better. However, I also found a bug in Sparse that could cause it to crash, whenever there is a declaration of a variable by specifying only "`typeof()`" as its type. The crash then happened after the lazy type evaluation of the declared variable (when the variable is used in the code).

When testing Sparse, I have noticed that it is possible to miss a very trivial uninitialized variable problem. I even discovered that the bug is also relevant to gcc, and that both of them loose the sight of the uninitialized variable due to optimizations. I have devised a simple addition to Sparse's optimization code that will make sure to check if the variable is initialized before optimizing it out. I have then ran the test on the Linux kernel and found out (among few false-positives) two real bugs that were not by Sparse, nor by gcc before.

Upon closer inspection of Sparse's context checking capabilities, I have discovered that the results are not very reliable. It is possible, however, that this is caused by a recent revert of very big part of context-related code.

As a way to help programmers to familiarize themselves with the Sparse library and the structure of the syntax tree, I wrote a simple utility that uses Sparse to parse a C source file and print out its syntax tree representation. If you are in doubts about the usage of Sparse data structures, refer to this program. I also tried to comment it as much as possible, where appropriate, although I believe that when one learns the basics of the syntax tree, he should be able to orientate fairly quickly by only using the header files provided by Sparse.

4.1.2 Mygcc

The Condate language parser that was written using Bison uses the `fgets()` function without making sure that the file stream is not at the end. For most of the errors that occur at the end of file, Mygcc will print out an uninitialized buffer, resulting in a garbage of characters. This can be reproduced for example, by passing file with the only word "condate" to Mygcc.

I also managed to find Mygcc behaving unexpectedly on numerous occasions, but did not manage to find a reproducer to the problem, and thus the cause is not known to me. Since Mygcc's declared status as an "experimental compiler", this probably should not be surprising.

4.2 Documentation

What I consider to be most useful from my work is the documentation of the Sparse library. I enjoyed reading through it and finding many quirks in the use of the C language that I did not even know were possible. I think it was a great experience, even though it consumed a lot of time. I was at first hoping that I could write a bigger back-end that would do a useful analysis. However, the documentation of the library is practically non-existent, except for a log of an IRC discussion with the Sparse maintainer, where he says few words about the data structures on a very high level.

4.3 Future directions

For the future, I would recommend a closer look at algorithms behind Mygcc. I especially hope that Mygcc, or some incarnation of it, will eventually find a way into gcc, or at least some other mainstream compiler. I really believe the vision of Mygcc's creator that using static code analysis techniques that are not really precise, but are very practical and easy to deploy, is the way for formal verification to become more popular.

Bibliography

[1] Aho, A. V.; et. al: *Compilers: Principles, Techniques, and Tools*. Addison Wesley, druhé vydání, Srpen 2006, ISBN 978-0321486813.

[2] Corbet, J.: Finding kernel problems automatically. Červen 2004, [online].
URL http://lwn.net/Articles/87538/

[3] Corbet, J.: Introducing gfp_t. Říjen 2005, [online].
URL http://lwn.net/Articles/155344/

[4] Corbet, J.: Using sparse for endianness verification. Říjen 2006, [online].
URL http://lwn.net/Articles/205624/

[5] Friedl, S.: Using GNU C __attribute__. Září 2002, [online].
URL http://unixwiz.net/techtips/gnu-c-attributes.html

[6] Harbison, S. P.; Steele, G. L.: *C: A Reference Manual*. Prentice Hall, páté vydání, Březen 2002, ISBN 978-0130895929.

[7] Kernighan, B. W.; Ritchie, D. M.: *The C Programming Language*. Prentice Hall, druhé vydání, Duben 1988, ISBN 978-0131103627.

[8] Stallman, R. M.; et. al.: *Using the GNU Compiler Collection (For GCC version 4.4.0)*. 1988-2008, [online].
URL http://gcc.gnu.org/onlinedocs/gcc-4.4.0/gcc.pdf

[9] Torvalds, L.: *Sparse README file*. Březen 2003.

[10] Torvalds, L.: Sparse "context" checking.. Říjen 2004, [online].
URL http://lwn.net/Articles/109066/

[11] Triplett, J.: Sparse – a Semantic Parser for C. [online], [rev. 2007-09-15], [cit. 2009-05-05].
URL http://www.kernel.org/pub/software/devel/sparse/

[12] Triplett, J.: *Sparse manual page*. Srpen 2007.

[13] Triplett, J.: *A slightly edited irc discussion with Josh Triplett*. Prosinec 2008.

[14] Volanschi, N.: mygcc. 2008, [online].
URL http://mygcc.free.fr/

[15] Volanschi, N.; Rinderknecht, C.: Unparsed Patterns: Easy User-Extensibility of Program Manipulation Tools (Extended Version). In *ACM SIGPLAN 2008 Workshop on Partial Evaluation and Program Manipulation (PEPM '08)*.
URL http://mypatterns.free.fr/unparsed/unparsed-pepm08-extended.pdf

[16] Volanschi, N.; Rinderknecht, C.: A portable compiler-integrated approach to permanent checking. *Automated Software Engineering*, ročník 15, č. 1, Březen 2008: s. 3–33, ISSN 0928-8910.
URL http://www.springerlink.com/content/t516646543w5117n/

Appendix A

Contents of the DVD

The DVD contains:

- `tex`: Directory with tex sources of this technical report.
- `src`: Source code and patches written during this work.
- `tests`: Test cases used to test Sparse checker and Mygcc.